

FLP Impossibility Theorem

Presented By Gilad Turok

Talk Overview

Outline:

- Background & Introduction
- System Model
- Proofs
 - Lemma 1
 - Lemma 2
 - Lemma 3
- Conclusion

Goal: minimize terminology

Background & Introduction

Impossibility of Distributed Consensus with One Faulty Process

MICHAEL J. FISCHER

Yale University, New Haven, Connecticut

NANCY A. LYNCH

Massachusetts Institute of Technology, Cambridge, Massachusetts

AND

MICHAEL S. PATERSON

University of Warwick, Coventry, England

Abstract. The consensus problem involves an asynchronous system of processes, some of which may be unreliable. The problem is for the reliable processes to agree on a binary value. In this paper, it is shown that every protocol for this problem has the possibility of nontermination, even with only one faulty process. By way of contrast, solutions are known for the synchronous case, the “Byzantine Generals” problem.

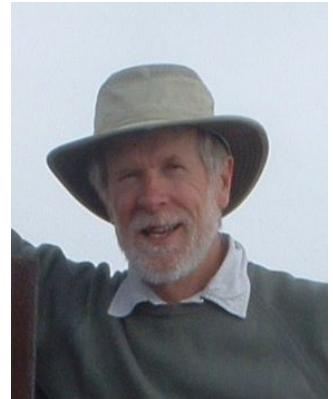
The Authors

Authors:

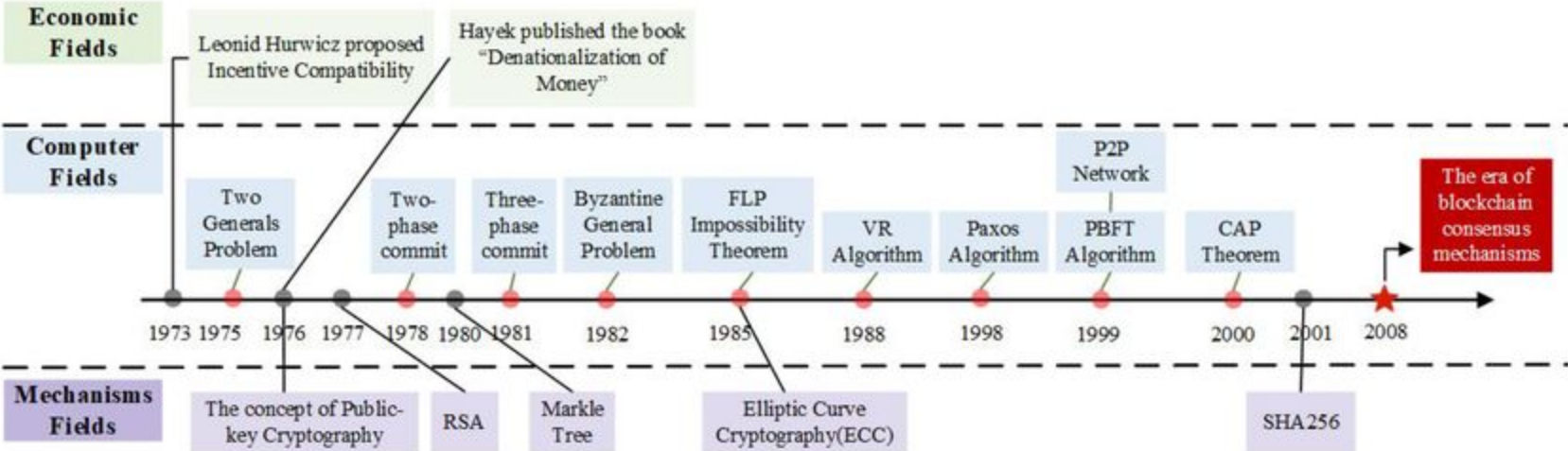
- Michael Fischer – Yale
- Nancy Lynch – MIT
- Michael Paterson – University of Warwick

Published: *Journal of the ACM*, April 1985

Impact: Dijkstra Prize (most influential paper in distributed computing)



Historical Timeline



Why Care About Consensus?

Problem: Getting distributed processes to agree on a value

Applications:


- Database transaction commits
- State machine replication
- Leader election
- Any coordination in distributed systems

Historical context: In 1980-1985, it was known consensus was solvable in synchronous systems. Open question: what about asynchronous systems?


Question: What's the minimal model where consensus is still possible?

Synchronous vs. Asynchronous

Synchronous Model:

- Shared global clock
- Bounded message delays (arrive within time step)
- Consensus solvable 

Asynchronous Model (Today):

- No shared clock
- Arbitrary (finite) message delays
- Only guarantee: eventual delivery
- Consensus solvable 

The FLP Result

Theorem (FLP 1985): No deterministic protocol solves consensus in the asynchronous model, even with:

- Only $f = 1$ faulty process
- Only **crash faults**

Crash fault: Process follows protocol correctly, then permanently stops (like pulling the plug).

Implication: Cannot guarantee both agreement AND termination under asynchrony.

System Model

What is a Process?

Each process p has:

- **Input:** Private initial value $x_p \in \{0,1\}$
- **State:** Internal state (evolves during execution)
- **Output:** Final decision $y_p \in \{b,0,1\}$ initialized to b (write-once)

Atomic process step:

1. **Receive** message m
2. **Process** message (update internal state based on current state + m)
3. **Send** messages to other processes

Key property: Steps 1-3 happen atomically (indivisibly) and deterministically given the process's current state and the received message.

Message Queue

Message pool M: Set of outstanding (not-yet-delivered) messages

Message format: (r, m)

- r = recipient process
- m = message payload (content)

Null messages: (r, \emptyset)

- \emptyset represents "no content" / empty message
- Used to ensure processes can participate even without "real" messages

The Asynchronous Model

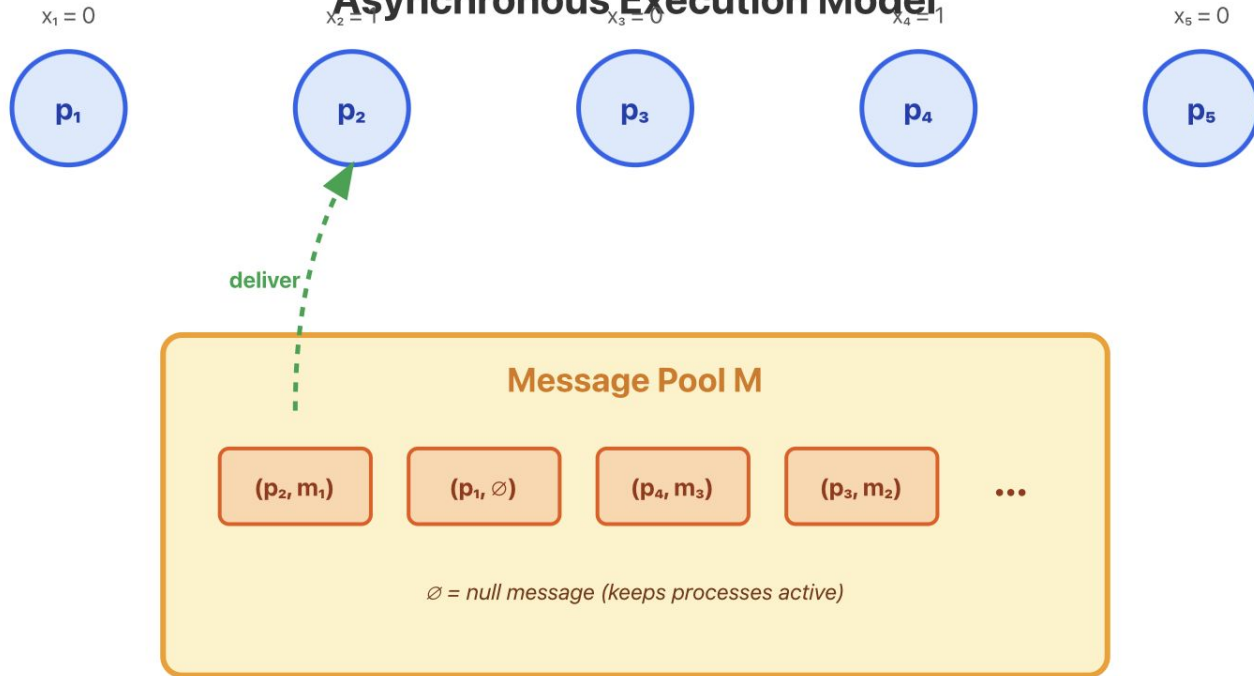
Execution:

```
while(TRUE) :  
    arbitrarily choose (r,m) ∈ M  
    remove (r,m) from M  
    deliver (r,m) to process r  
    process r updates internal state based on message m  
    process r adds new messages to M (optional)
```

Key properties:

- **No shared clock:** Processes don't know "when" events happen
- **Arbitrary delays:** Messages can take arbitrarily long (but finite)
- **Eventual delivery:** Every message in M is eventually delivered

Asynchronous Execution Model



Protocol Definition

What is a protocol? A protocol P specifies what each process does when it receives a message.

Process p 's behavior determined by:

1. Its private input $x_p \in \{0,1\}$
2. Sequence of messages it has received: m_1, m_2, \dots, m_p

Deterministic: Given same $(x_p, \text{message sequence})$, process behaves identically.

Local view constraint: Process p only knows:

- Its own input x_p + messages it has received
- NOT: other processes' inputs, states, or messages

The Consensus Problem

Goal: Design a protocol P where all non-faulty processes output same value for y_i

Setup:

- n processes, each with private input $x_i \in \{0,1\}$
- At most f processes may crash

Three required properties:

1. **Termination:** Every non-faulty process eventually outputs $y_i \in \{0,1\}$
2. **Agreement:** All non-faulty processes output same value
3. **Validity:** If all non-faulty processes have $x_i = v^*$, then all output $y_i = v^*$

Discussion #1

- Why is validity necessary? What happens if we only require termination + agreement?
- What about termination + validity only?

Configurations

Configuration C: Complete snapshot of protocol state

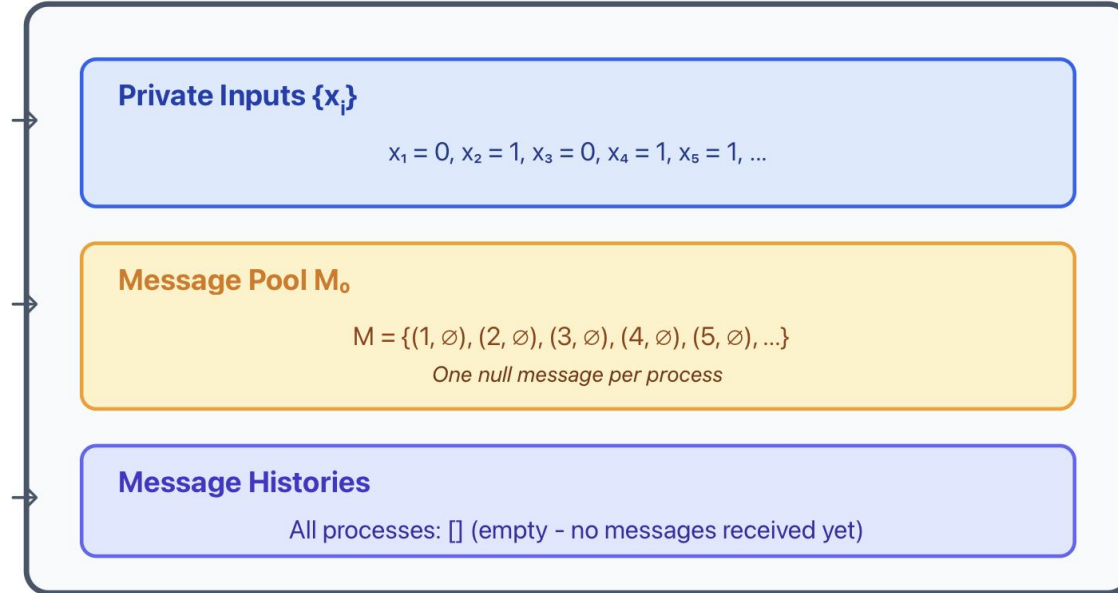
- Message pool M
- Private inputs $\{v_i\}$
- Message sequences received by each node

Initial Configuration

- Each process p has private input $x_p \in \{0, 1\}$ and output $y_p = b$
- Message queue $M = \{(1, \emptyset), (2, \emptyset), \dots, (n, \emptyset)\}$
- No messages received yet by any process

Initial Configuration C_0

Starting point of protocol execution



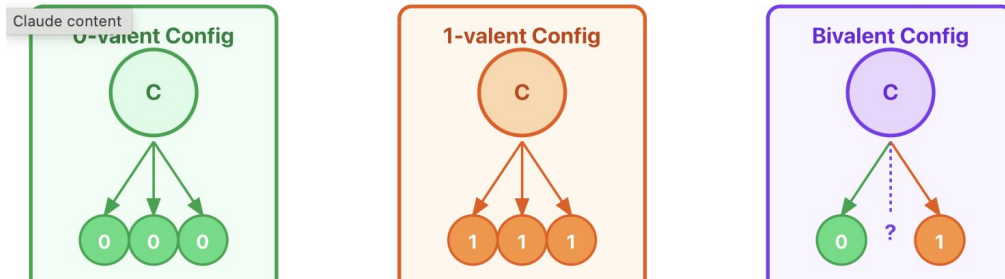
Configuration Types (Valency)

Definitions:

- **0-valent configuration:** All possible executions \rightarrow all decide 0
- **1-valent configuration:** All possible executions \rightarrow all decide 1

More Definitions:

- **Univalent:** configuration is 0-valent or 1-valent (value *determined*)
- **Bivalent configuration:** configuration can decide 0 or 1 (value *undetermined*)



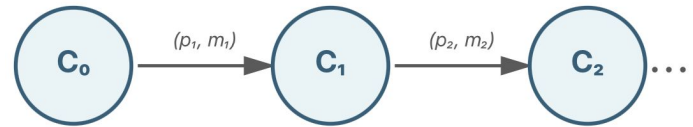
Events and Configuration Graphs

Event $e = (p, m)$: The atomic action of delivering message m to process p

- Causes process p to: receive m , update state, send new messages
- Transforms one configuration to another: $C \xrightarrow{e} C'$ or $C \xrightarrow{(p,m)} C'$
- Deterministic: same event from same config always produces same result

Configuration graph:

- Vertices = configurations
- Edges = events (message deliveries)



Protocol execution = path through configuration space

The Proofs 

Proof Strategy

Goal: Show protocol P cannot satisfy all three properties.

Approach: Construct infinite run (violates termination) that satisfies agreement and validity.

Three-step construction:

1. **Lemma 1:** Commutativity of independent events
2. **Lemma 2:** \exists initial bivalent configuration C_0
3. **Lemma 3:** Can extend bivalent sequence: $C_i \rightarrow C_{i+1}$

Result: Infinite sequence $C_0 \rightarrow C_1 \rightarrow C_2 \rightarrow \dots$ of bivalent configs contradicts termination.

Lemma 1: Commutativity

Lemma 1: If two events $e_1 = (p, m)$ and $e_2 = (p', m')$ apply to configuration C and affect different processes ($p \neq p'$), then they commute:

$$e_1(e_2(C)) = e_2(e_1(C))$$

Intuition: If processes p and p' don't interact in these steps, order doesn't matter.

Why it matters: Used in Lemma 3 to show certain message orderings lead to same configuration.

Lemma 2: Initial Bivalency

Lemma 2: For any protocol P satisfying agreement and validity, there exists a choice of private inputs where the initial configuration C_0 is bivalent.

Proof strategy: By contradiction - assume all initial configurations are univalent.

Setup: Consider $n+1$ initial configurations:

- X_0 : all inputs = 0 [validity \Rightarrow 0-valent]
- X_1 : input 1 is 1, rest are 0
- X_2 : inputs 1,2 are 1, rest are 0
- ...
- X_n : all inputs = 1 [validity \Rightarrow 1-valent]

Lemma 2: Proof

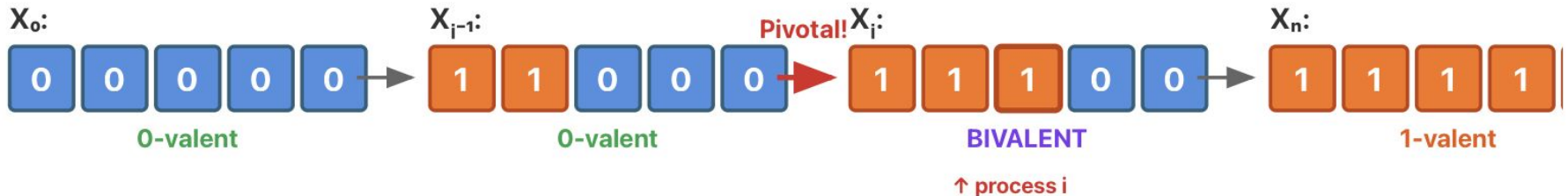
Proof by contradiction: Assume all initial configurations are univalent.

Key observation: X_0 is 0-valent (validity), X_n is 1-valent (validity).

Let i = smallest index where X_i is not 0-valent.

Must have: X_i is 1-valent (by our assumption - all configs are univalent).

So: Adjacent configs X_{i-1} (0-valent) and X_i (1-valent) differ only in process i 's input.



Lemma 2: Proof

We have: X_{i-1} is 0-valent, X_i is 1-valent, differing only in process i 's input.

Key: Suppose process i crashes (fails) at the start.

From X_i with i crashed:

- Other processes never hear from i
- Execution is indistinguishable from X_{i-1} (where i has input 0)
- Must decide 0 (since X_{i-1} is 0-valent)

From X_i with i not crashed:

- Must decide 1 (since X_i is 1-valent)

Contradiction: Same config X_i leads to both outcomes depending on whether i crashes

Therefore, X_i must be **bivalent**. ■

Discussion #2

In Lemma 2, why does it matter that we only need $f = 1$ faulty process?

How does the crash fault model compare to Byzantine faults? Why is proving impossibility for crash faults more impressive?

Lemma 3: Extending Bivalency

Lemma 3: Let C be a bivalent configuration and $e = (p, m)$ an event applicable to C . Then C has a bivalent successor in which e has been applied.

Intuition: From bivalent C , we can apply e (eventually) while staying bivalent.

Why needed: Combined with Lemma 2, builds infinite bivalent sequence where every message is eventually delivered.

Lemma 3: Proof Setup

Define: Let $\mathcal{E} = \{e(D) \mid D \text{ reachable from } C \text{ without applying } e\}$

In words: \mathcal{E} is the set of all configurations where:

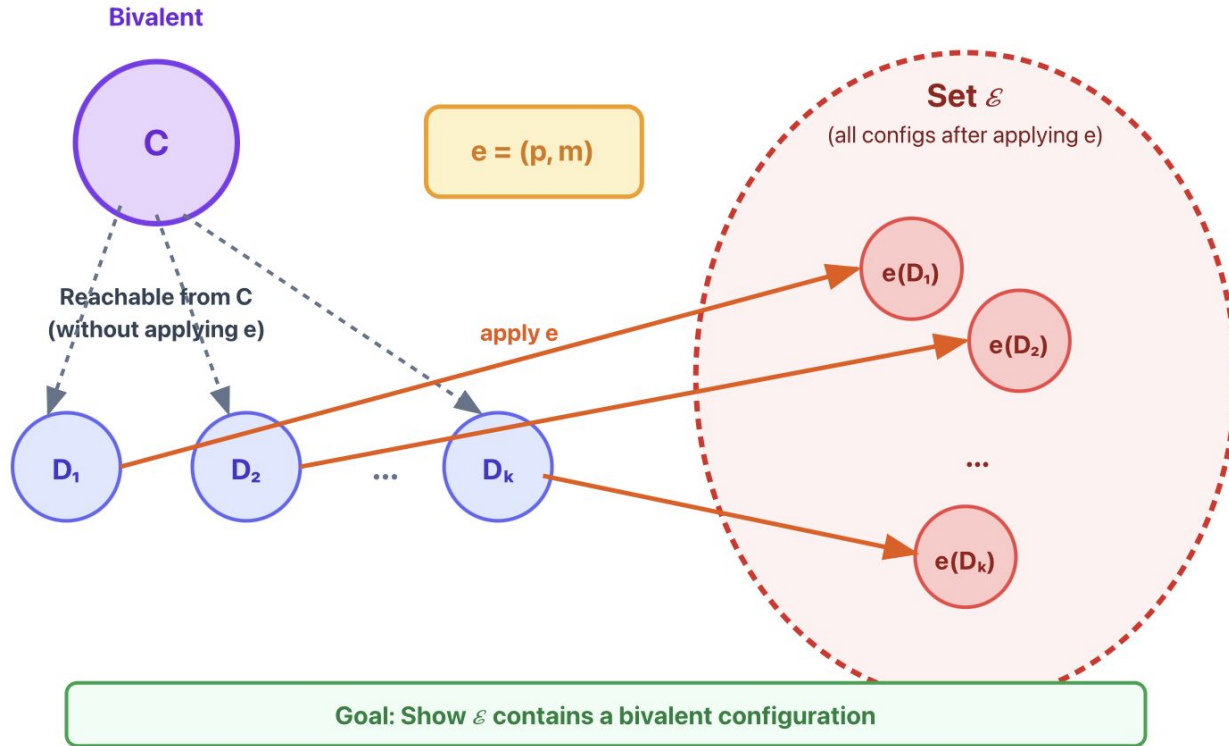
1. Start from C
2. Deliver some sequence of messages (not including e)
3. Then deliver e

Goal: Show \mathcal{E} contains a bivalent configuration.

Proof strategy: Assume all configs in \mathcal{E} are univalent \rightarrow derive contradiction.

Lemma 3: Proof Setup

Define set $\mathcal{E} = \{e(D) \mid D \text{ reachable from } C \text{ without applying } e\}$



Lemma 3: Finding 0-valent and 1-valent configs

Since C is bivalent, there exist:

- A run from C leading to all-zero outcome
- A run from C leading to all-one outcome

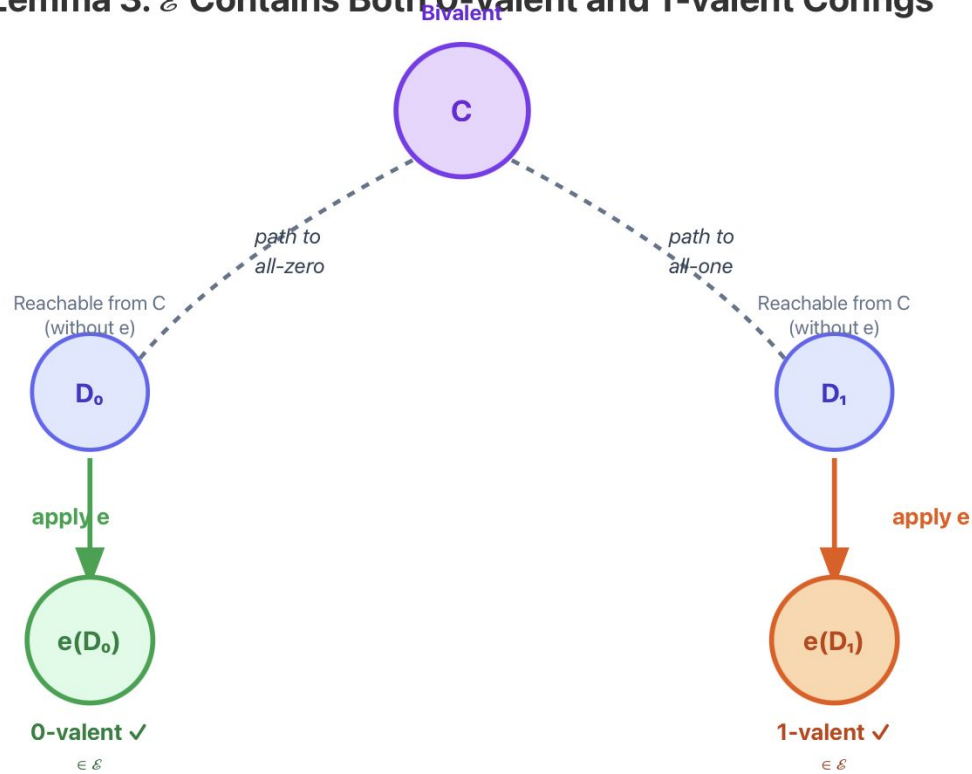
By eventual delivery: Message e must be delivered in both runs.

Therefore \exists configurations D_0, D_1 reachable from C (without e) where:

- $e(D_0)$ is 0-valent (applying $e \rightarrow$ 0-valent config)
- $e(D_1)$ is 1-valent (applying $e \rightarrow$ 1-valent config)

Both $e(D_0)$ and $e(D_1)$ are in \mathcal{E} .

Lemma 3: \mathcal{E} Contains Both 0-valent and 1-valent Configs



Why: **C** is bivalent, so paths exist to both outcomes. Event **e** must be applied eventually (eventual delivery).

Lemma 3: The Pivotal Transition

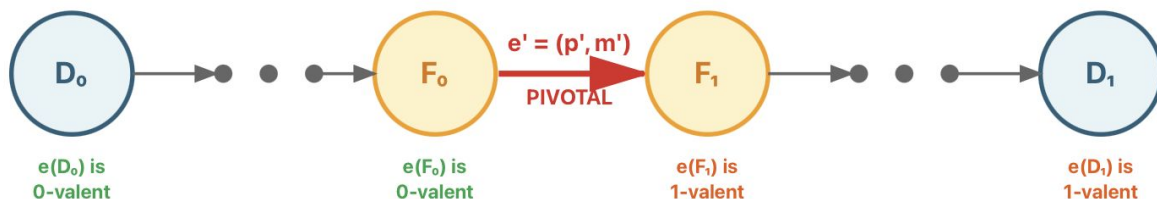
Key observation: There's a path $D_0 \rightarrow D_1$ (sequence of events not including e).

Along this path, find adjacent configurations F_0, F_1 where:

- $F_1 = e'(F_0)$ for some event $e' = (p', m')$
- $e(F_0)$ is 0-valent
- $e(F_1)$ is 1-valent

This transition exists because:

- $e(D_0)$ is 0-valent (starting point)
- $e(D_1)$ is 1-valent (ending point)
- Must cross from 0-valent to 1-valent somewhere



Path consists of events NOT including $e = (p, m)$

Lemma 3: Case 1 ($p \neq p'$)

Case 1: Events e and e' apply to different processes.

By Lemma 1 (commutativity): $e(e'(F_0)) = e'(e(F_0))$

Therefore: $e(F_1) = e(e'(F_0)) = e'(e(F_0))$

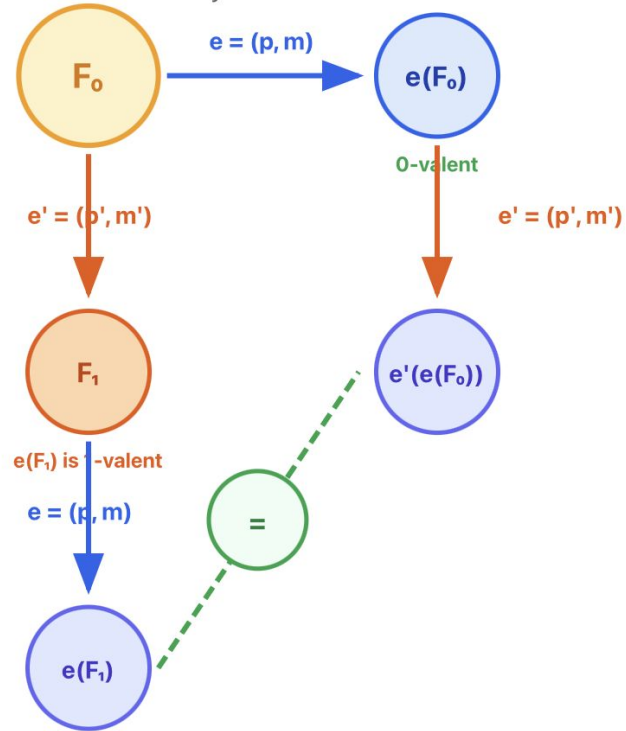
But:

- $e(F_0)$ is 0-valent (by our choice of F_0)
- $e'(e(F_0))$ is also 0-valent (once univalent, stays univalent)
- So $e(F_1)$ is 0-valent

Contradiction: We said $e(F_1)$ is 1-valent!

Lemma 3 Case 1: Events to Different Processes

If $p \neq p'$, then e and e' commute by Lemma 1



CONTRADICTION: $e(F_1) = e'(e(F_0))$ must be 0-valent, but we said $e(F_1)$ is 1-valent!

Lemma 3: Case 2 ($p = p'$)

Case 2: Both $e = (p, m)$ and $e' = (p, m')$ go to same process p .

Consider execution from $e(F_0)$ where p takes no further steps (crashes immediately):

- Other processes must eventually decide (termination)
- Let v_0 be the decision value
- This decision depends only on:
 - Other processes' states
 - Messages from processes $\neq p$
 - The fact that p is silent

Lemma 3: Case 2 Continued ($p = p'$)

Now consider execution from $e(F_1) = e(e'(F_0))$ where p crashes immediately:

- Other processes see:
 - Same initial states
 - Same messages from processes $\neq p$
 - Same silence from p (crashed after initial event)
- Must decide same value v_0

But:

- $e(F_0)$ is 0-valent, so $v_0 = 0$
- $e(F_1)$ is 1-valent, so must lead to $v_1 = 1$

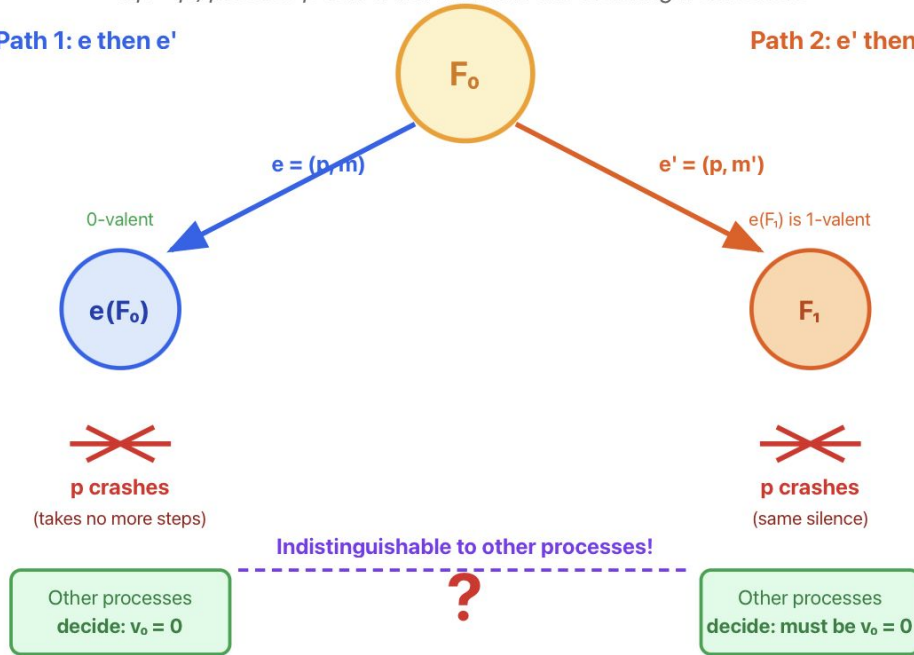
Contradiction!

Lemma 3 Case 2: Events to Same Process

If $p = p'$, process p can crash and hide the ordering from others

Path 1: e then e'

Path 2: e' then e



CONTRADICTION: Can't have $v_0 = 0$ in both paths when $e(F_1)$ is 1-valent!

Putting It Together

Theorem (FLP): No deterministic protocol solves consensus with $f=1$ crash fault in asynchronous model.

Proof:

1. Start with initial bivalent C_0 (Lemma 2)
2. At each C_i , let (r_i, m_i) = oldest message in pool
3. Apply Lemma 3 to find bivalent C_{i+1} where (r_i, m_i) is delivered

Result: Infinite sequence $C_0 \rightarrow C_1 \rightarrow C_2 \rightarrow \dots$ where:

- All C_i are bivalent (protocol never decides)
- Every message eventually delivered (satisfies model)

Contradicts termination. ■

Conclusion

Implications

What FLP teaches us:

1. **Agreement vs. Termination tradeoff:** Can't guarantee both under asynchrony
 - BFT protocols (Tendermint): favor agreement
 - Longest-chain (Bitcoin): favor termination
2. **Assumptions matter:** Synchrony is not just convenience—it's fundamental
3. **Guides design:** Points toward "sweet spot" models

Discussion #3

The FLP result shows consensus is impossible in the asynchronous model with even one crash fault. What are possible ways around this impossibility?

What assumptions could we add or properties could we weaken?

Escaping FLP

Three main approaches:

1. **Randomization**: Probabilistic termination guarantees
2. **Partial synchrony**: Assume eventually synchronous
3. **Weaken termination**: Finite expected time, high probability bounds

Key insight: No "free lunch"—must compromise somewhere.

References

Original paper: Fischer, M.J., Lynch, N.A., Paterson, M.S. (1985). "Impossibility of Distributed Consensus with One Faulty Process." *Journal of the ACM*, 32(2):374-382.

This presentation based on: Roughgarden, T. (2021). "Foundations of Blockchains: Lectures #4 & 5."